Machine Learning Exercise 3 Report

COMP24112

X83005SZ

L2 Regularized Least Squares

My implementation of the l2 regularized least squares approach is simple. The main idea behind this exercise is that we want to minimize the sum of squares loss- this can be done by trying to minimize the gradient of the polynomial sum of squares error. The optimal solution of this problem is found by setting the gradient to zero, and then rewriting this normal equation in terms of X and y, to solve for your optimal weights. Our version has a regularization term included, which allows us to take the inverse of a matrix that might not otherwise be invertible, which is a necessary step for calculating the optimal weights.

My design first initializes some necessary variables for the weight optimizations, such as X tilde. After this, it checks the hyperparameter used- if the regularization parameter is equal to zero, the pseudoinverse function is used in place of the standard inverse function for the calculation. For lambda > 0, the function then checks if there are more rows or columns- if there are more rows than columns, then the system is overdetermined, and a unique solution exists. If the number of rows is less than or equal to the number of columns, then there exist infinite solutions to the system of equations, and the optimal weights are calculated in a slightly different way. Note that because this part of the code is only reachable if lambda > 0, all X^T @ X and X @ X^T are invertible after the regularization term is added.

Experiment 1

The first step of classification was preparing my test and training data. For my training data, I choose 3 samples from each class, and then filter this set so that only data from class 1 and 30 is included in both test and training sets. I then train the model using this filtered dataset using 0 for hyperparameter lambda, meaning that the weights are calculated using the pseudoinverse. After this, I use the predict function to predict the class for each element in my test class. I set different threshold values for each iteration of the experiment, basically acting as a cutoff point for the classification- if it is above this value and the actual class is 30, it is classified as correct, likewise if the predicted output is below this threshold value and the actual class is 1. Otherwise, the classification is marked as incorrect.

Changing the class labels only minorly affected my model performance. The best performing set of class labels were (-1,1) with a threshold value of 0, followed by (0,1) with threshold of .5, and finally (1,30) with a threshold value of 15.5. My mean training accuracies for these experiment batches were all 100%, but the testing errors for all were around 6.5%. The class labels should not have any impact on the performance of the model, if for binary classification, the data is split evenly, and the threshold is halfway between the class labels.

Experiment 3

My absolute percentage errors for both training and testing of my face completion model were both relatively small- nearly 0 and 11%, respectively. I believe that most of the testing error comes from faces with facial hair, faces at an angle, and faces not smiling. If you look at the completed images created by the model, often it tries to give the right side of the face a slight smile, likely because some faces are smiling, and some aren't in the training set. When faces are tilted at an angle to the camera, the dimensions of the face are often messed up- because most of the faces are looking directly at the camera, the model does not adjust the dimensions to account for the tilted angle very well, and often the right side of the face looks stretched compared to the left side of the face. I think that the model can be improved through more training, especially with faces with more unique features such as facial hair, classes, tiling faces, and different expressions. The faces would also look better if more effort was made to match the pixel color of the left side of the face- even if the facial features are spot on, often the skin tone is strikingly different between sides of the face.

Gradient Descent Learning Rate and Iterations

I ended up using a learning rate of 10^-3 for my gradient descent function. By increasing the learning rate to 10^-2, the learning rate was much too high, causing a memory overflow when run. When the learning rate was 10^-4, the model performed well, but not optimally- the model reaches its peak accuracy after about 25 weight updates with learning rate of 10^-4, compared to less than 15 iterations with 10^-3. I chose the iteration number of 200 because after examining my graphs for different costs, I was unable to discern much change in cost after this point. While the training accuracy is stable early in the process, the cost continues to decrease as more iterations happen.

Comparing Gradient Descent with Stochastic Gradient Descent

For this experiment, I train and predict with each model 50 times, with 200 weight updates each iteration for each model. I use the parameters that gave me the best performance from both models to allow each model to perform as best as it can, given my implementation. Each iteration, the model selects 3 new random samples from 2 classes, 1 and 30, and cleans the data like previous experiments. I use class labels –1 and 1 for classes 1 and 30, and I use a threshold value of 0 for my classification. I compare the average accuracy measured at each iteration of the process for each function, the cost at each iteration, and the average time to train the model.

The plot of average training accuracy for the gradient descent has a nearly immediate jump from 50% to 100%, as should be expected for a model like this. However, the stochastic gradient descent training accuracy is extremely unstable- while it has a minor upward trend, this is very slight, with the maximum average training accuracy around 64%. The same is true of the testing accuracy for both models- the gradient descent model almost immediately stabilizes its testing accuracy at around 90%, while the stochastic gradient descent, with a slight upward but unstable upward trend, maximizes at around 56%.

The instability of the stochastic gradient descent, especially compared to the traditional gradient descent method, is to be expected. The SGD method retrains the weights based on the gradient given one randomly selected input vector, instead of using all the input vectors as the GD method uses. The simpler design of the SGD method is reflected in the runtime- it averages .2 seconds to train the stochastic model, while it averages 1.1 seconds to train the traditional GD model. The cost over time does tend to decrease for the SGD method, but not my much meaningful. I believe that with increased iterations, the SGD model might perform better, but at a lower iteration number (around 200), the GD model is better in all aspects but performance.